INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Finite Eilenberg Machines

Benoît Razet

N° 6486

February 2008

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6486--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Finite Eilenberg Machines

Benoît Razet

Thème SYM — Systèmes symboliques
Équipe-Projet SANSKRIT

Rapport de recherche n° 6486 — February 2008 — 17 pages

Abstract: Eilenberg machines define a general computational model. They are well suited to the simulation of problems specified using finite state formalisms such as formal languages and automata theory. This paper introduces a subclass of them called finite Eilenberg machines. We give a formal description of complete and efficient algorithms which permit the simulation of such machines. We show that our finiteness restriction ensures a correct behavior of the simulation. Interpretations of this restriction are studied for the particular cases of non-deterministic automata (NFA) and rational transducers, leading to applications to computational linguistics. The given implementation provides a generic simulation procedure for any problem encoded as a composition of finite Eilenberg machines.

Key-words: Automata, formal languages, Eilenberg machines

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Finite Eilenberg Machines

Résumé : Les machines d'Eilenberg définissent un modèle de calcul général. Elles permettent de spécifier et résoudre des problèmes de la théorie des langages formels. Ce travail introduit une sous-classe de ces machines qu'on appelle les machines d'Eilenberg finies. De telles machines se simulent à l'aide d'un algorithme effectif pour lequel une spécification formelle est fournie. Nous prouvons que le critère de finitude assure le bon comportement de l'algorithme. Nous discutons aussi de l'impact de ce critère dans le cas où de telles machines sont en fait des automates ou des transducteurs rationnels. Ceci légitime cette étude dans le cadre de la linguistique computationnelle.

Mots-clés : Automates, langages formels, machines d'Eilenberg

Introduction

Samuel Eilenberg introduced in the chapter 10 of his book [4], published in 1974, a notion of *Machine* which he claimed to be a *very efficient tool* for studying formal languages of the Chomsky hierarchy. They are sometimes referred to as *X-machines*. Many variants have appeared in the last twenty years [1] in several scientific domains different from formal languages.

Eilenberg machines define a general computational model. Assumed given an abstract data set X (it motivates X-machine terminology), a *machine* is defined as an automaton labelled with binary relations on X. Two generalizations result from this. Firstly, the set X abstracts the traditional tape used by automata on words, transducers *etc.* Secondly, compared to functions, binary relations give a built-in notion of non-determinism. Many translations of other machines into Eilenberg machines are given in [4]: automata, transducers, real-time transducers, two-way automata, push-down automata and Turing machines.

The remainder of this paper recalls the definitions of Eilenberg machines in Section 1. It also introduces a new subclass of them called *finite Eilenberg machines*. The Section 2 motivates their utility discussing two examples. Firstly, we show that any non-deterministic finite automaton (NFA) may be encoded as a finite Eilenberg machine solving its word problem. Secondly, we discuss rational relations and transducers providing encodings into machines solving the three problems of recognition, synthesis and analysis. Resulting machines might not satisfy our finiteness restriction. We discuss the ones for which it is the case considering *length-preserving relations* and furthermore a variant of *positive rational relations*. Applications are also motivated by computational linguistics. The next two sections provide algorithms simulating finite Eilenberg machines. Since relations are central to the considered machines, the Section 3 proposes an encoding of them using streams. The Section 4 gives a formal description of algorithms which permit the simulation of finite Eilenberg machines in the spirit of the *reactive engine* introduced in [5]. Formal proofs are given ensuring the simulation is correct.

1 Finite Eilenberg machines

We consider a monoid with carrier M, $\cdot$ an associative product on M and 1 its unit element. A finite monoid automaton $\mathcal{A}$ over M, also called a M-automaton, is a tuple (Q, δ, I, T) with Q a finite set of elements called *states*, δ a function from Q to finite subsets of $(M \times Q)$ called the *transition function*, I a subset of Q of *initial states* and T a subset of Q of *terminal states*.
A *path* p is a sequence $p = q_0 \xrightarrow{m_1} q_1 \xrightarrow{m_2} \cdots \xrightarrow{m_n} q_n$ with $n \in \mathbb{N}$ and $\forall i \leq n\ q_i \in Q$, $\forall i < n\ m_{i+1} \in M$ and $\forall i < n\ (m_{i+1},\ q_{i+1}) \in \delta(q_i)$. The path p is *successful* when $q_0 \in I$ and $q_n \in T$; its *length* is n. The *label* of p written $\bar{p}$ is 1 if $n = 0$ or $m_1 \cdot \ldots \cdot m_n$ otherwise. Finally, the *behavior* of the M-automaton $\mathcal{A}$, written $|\mathcal{A}|$, is defined as the set of all labels of successful paths of $\mathcal{A}$. We introduce the

type of $\mathcal{A}$, written $\Phi_{\mathcal{A}}$, as the finite subset of M of elements appearing in the image of δ:

$$\Phi_{\mathcal{A}} = \{\ m \in M \mid \exists\ q\ q' \in Q,\ (m, q') \in \delta(q)\ \}\ .$$

Let us now precise some notations for relations which are central to the remainder of this paper. A relation ρ from some set $\mathcal{D}$ to a set $\mathcal{D}'$ written $\rho : \mathcal{D} \to \mathcal{D}'$ is a set of pairs from $\wp(\mathcal{D} \times \mathcal{D}')$. The functional notation of its type is justified by the isomorphism between $\wp(\mathcal{D} \times \mathcal{D}')$ and $\mathcal{D} \to \wp(\mathcal{D}')$. The converse of a relation $\rho : \mathcal{D} \to \mathcal{D}'$ is written $\rho^{-1} : \mathcal{D}' \to \mathcal{D}$. Let us use $\rho(d)$ as notation for $\{\ d' \mid d' \in \mathcal{D}',\ (d, d') \in \rho\}$.

Let us recall Eilenberg's definition of machines.
Let $\mathcal{D}$ be an arbitrary set called the *data* (it replaces the original notation X). We consider the set $R_{\mathcal{D}}$ of binary relations from $\mathcal{D}$ to $\mathcal{D}$. We consider here the *relations monoid* $R_{\mathcal{D}}$ with the *relation composition* $\circ$ as associative product and the *identity* relation *id* as unit element. A $\mathcal{D}$*-machine* $\mathcal{M}$ is a $R_{\mathcal{D}}$-automaton (Q, δ, I, T). With respect to the previous definitions the label of a path $p = q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \cdots \xrightarrow{\phi_n} q_n$ is the composition of relations $\bar{p} = \phi_1 \circ \cdots \circ \phi_n$. The behavior of $\mathcal{M}$ as an automaton, $|\mathcal{M}|$, is the set of relations of all labels of successful paths. The distinction between an automaton and a machine lies in the use of the union operation available on relations. The machine $\mathcal{M}$ defines a particular relation, written $||\mathcal{M}||$, as the relation union extended over all relations in $|\mathcal{M}|$:

$$||\mathcal{M}|| = \bigcup_{\rho\, \in\, |\mathcal{M}|} \rho\ .$$

We call the relation $||\mathcal{M}||$ the *characteristic relation* of the machine $\mathcal{M}$. We have given until now what we call the *kernel* of an Eilenberg machine which refers only to the automaton part.

The complete description of an Eilenberg machine requires what we call its *interface*. That is, consider $\mathcal{D}_-$ and $\mathcal{D}_+$ be two sets called respectively the *input* and *output* sets, an input relation $\phi_- : \mathcal{D}_- \to \mathcal{D}$ and an output relation $\phi_+ : \mathcal{D} \to \mathcal{D}_+$. Intuitively, the relation ϕ_- feeds the kernel with inputs and ϕ_+ interprets kernel results as outputs. A machine kernel with its interface defines a relation $\rho : \mathcal{D}_- \to \mathcal{D}_+$ as

$$\rho = \phi_- \circ ||\mathcal{M}|| \circ \phi_+\ .$$

The usefulness of kernel and interfaces will be clear with examples provided in section 2.

Remark 1.1 (on modularity of Eilenberg machines)**.** *Any Eilenberg machine* $\mathcal{M}$ *of type* $\Phi_{\mathcal{M}}$ *defines a characteristic relation* $||\mathcal{M}||$ *that may belong to a type* $\Phi_{\mathcal{M}'}$ *of another Eilenberg machine* $\mathcal{M}'$*. This gives an idea that Eilenberg machines describe a modular computational model.*

We are now going to introduce a new subclass of Eilenberg machines. For this purpose let us first define useful notions specific to machines rather than

automata. Let us consider a $\mathcal{D}$-machine $\mathcal{M} = (Q, \delta, I, T)$. We call *cell* a pair $c = (d, q)$ of $\mathcal{D} \times Q$. An *edge* is a triple $((d, q), \phi, (d', q'))$, written $(d, q) \xrightarrow{\phi} (d', q')$ and satisfying the two following conditions $(\phi, q') \in \delta(q)$ and $d' \in \phi(d)$. A *trace* is a sequence of consecutive edges $t = c_0 \xrightarrow{\phi_1} c_1 \cdots \xrightarrow{\phi_n} c_n$. The integer n is the *length* of the trace. The cell c_0 is called its *beginning* and c_n its *end*. For each data d and state q, the cell (d, q) defines a null trace with itself as beginning and end. A cell (d, q) is said to be *terminal* whenever q is terminal. A trace t is said to be *terminal* when its end is terminal. Remark that each trace can be projected as the corresponding path when data are forgotten.

Definition 1.1.

1. *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two sets, we say that a relation $\rho : \mathcal{D}_1 \to \mathcal{D}_2$ is locally finite iff for all data d in $\mathcal{D}_1$ the set $\rho(d)$ is finite.*
2. *We say that a machine $\mathcal{M}$ is locally finite iff every relation ϕ in $\Phi_{\mathcal{M}}$ is locally finite.*
3. *The machine $\mathcal{M}$ is globally finite iff its characteristic relation $||\mathcal{M}||$ is locally finite.*
4. *The machine $\mathcal{M}$ is nœtherian iff there is no infinite trace*

$$c_0 \xrightarrow{\phi_1} c_1 \ \cdots \ \xrightarrow{\phi_n} c_n \ \cdots.$$

5. *The machine $\mathcal{M}$ is called finite iff it is locally finite and nœtherian.*

Remark 1.2. *A locally finite machine may or may not be globally finite and conversely a globally finite machine may or may not be locally finite.*

Proposition 1.1. *If the machine $\mathcal{M}$ is finite then it is globally finite.*

Proof. Using König's lemma; the locally finite condition corresponds to the finite branching condition and the nœtherian condition to the non existence of infinite traces. □

Corollary 1.1. *Let ϕ_- and ϕ_+ be two partial functions. If the machine $\mathcal{M}$ is finite with interface ϕ_- and ϕ_+ then the relation $\phi_- \circ ||\mathcal{M}|| \circ \phi_+$ is locally finite.* □

We are interested in such finite Eilenberg machines because we show in section 4 that their simulation may be implemented in a complete fashion taking advantage of the finiteness and nœtherian properties. Before that, we discuss examples and applications of finite Eilenberg machines.

2 Examples and applications

Examples from Eilenberg showing his machine model implements many others used *"relabelings"* which are formally presented in [10]:

Definition 2.1. *Let M_1 and M_2 be two monoids, $\alpha : M_1 \to M_2$ a monoid morphism and $\mathcal{A}_1 = (Q, \delta_1, I, T)$ a M_1-automaton. A M_2-automaton $\mathcal{A}_2 = (Q, \delta_2, I, T)$ is a relabelling of $\mathcal{A}_1$ by α iff $\mathcal{A}_2$ is a copy of $\mathcal{A}_1$ except that each label m in the image of δ_1 is replaced by $\alpha(m)$ in δ_2. That is $\Phi_{\mathcal{A}_2} = \alpha(\Phi_{\mathcal{A}_1})$, the morphism α being extended to sets.*

The main result on relabellings is the following:

Lemma 2.1. *Let $\alpha : M_1 \to M_2$ be a monoid morphism, $\mathcal{A}_1$ a M_1-automaton and $\mathcal{A}_2$ a M_2-automaton relabelled from $\mathcal{A}_1$ by α. Then the behavior of $\mathcal{A}_2$ is the image by α of the behavior of $\mathcal{A}_1$, i.e.:*

$$|\mathcal{A}_2| = \alpha(|\mathcal{A}_1|)$$

Proof. Classic *e.g.* [10]. □

Eilenberg provided relabellings that translate well-known finite state formalisms into machines. We are going to discuss cases when the resulting machines are in fact finite Eilenberg machines. For this purpose let us recall some notions. We consider a finite set Σ of *letters* called the *alphabet*. We consider the free monoid Σ^* of *words* over Σ with the word concatenation as monoid product and the *empty word* ϵ as unit element. Formal languages are sets of words. Four basic operations on words are to be considered for defining Eilenberg machines for the next examples. For each letter σ of Σ:

- $L_\sigma = \{ (w, \sigma w) \mid w \in \Sigma^* \}$
- $R_\sigma = \{ (w, w\sigma) \mid w \in \Sigma^* \}$
- $L_\sigma^{-1} = \{ (\sigma w, w) \mid w \in \Sigma^* \}$
- $R_\sigma^{-1} = \{ (w\sigma, w) \mid w \in \Sigma^* \}$

The L and R denotations indicate operations respectively on the *left* or the *right* of a word. The last two relations are respectively the converse relations of the first ones. We will also use the *identity relation* written id_{Σ^*} as the relation $\{ (w, w) \mid w \in \Sigma^* \}$.

Remark 2.1. *Relations L_σ, R_σ, L_σ^{-1} and R_σ^{-1} described above are in fact partial functions, thus they are locally finite relations.*

2.1 NFA and finite Eilenberg machines

We consider here an alphabet Σ and words as elements of Σ^*. A NFA on alphabet Σ is a Σ^*-automaton $\mathcal{A}$ such that $\Phi_\mathcal{A} \subseteq \Sigma$ (ϵ-transitions are not

allowed). The set of words $|\mathcal{A}|$, the behavior of $\mathcal{A}$, is a formal language that belongs to the class of rational languages.

Let us define a relabelling procedure translating any NFA into an Eilenberg machine solving its word problem. Let $\mathcal{A} = (Q, \delta, I, T)$ be a NFA. We choose a data set $\mathcal{D} = \Sigma^*$. Since $\Phi_{\mathcal{A}} \subseteq \Sigma$, we define our morphism α on each element of Σ as $\alpha(\sigma) = L_\sigma^{-1}$. Thus the machine $\mathcal{M}$ relabelled from $\mathcal{A}$ by α has the following characteristic relation: $||\mathcal{M}|| = \{\ (w\,w',\ w')\ |\ w \in |\mathcal{A}|,\ w' \in \Sigma^*\ \}$. That is, a given input $w \cdot w'$ is truncated by the word w recognized by the automaton $\mathcal{A}$. The machine $\mathcal{M}$ is a finite Eilenberg machine because it satisfies the locally finite condition due to remark 2.1 and the nœtherian condition because for every edge $(w, q) \xrightarrow{L_\sigma^{-1}} (w', q')$ we have $|w'| = |w| - 1$, this shows that the length of traces is bounded by the length of their beginning word in their initial cell and thus there may not be infinite traces.

The encoding is complete when considered this finite Eilenberg machine with the following interface: $\mathcal{D}_- = \Sigma^*$, the input relation $\phi_- = id_{\Sigma^*}$, $\mathcal{D}_+ = \mathbb{B}$ the Boolean set composed of the two values $\top$ and $\bot$ and the output function $\phi_+ : \Sigma^* \to \mathbb{B}$ defined by $\phi_+(w)$ being $\top$ when $w = \epsilon$ and $\bot$ otherwise. Now we have $(\phi_- \circ ||\mathcal{M}|| \circ \phi_+)^{-1}(\top) = |\mathcal{A}|$. It shows that the relabelling is correct.

Remark 2.2. *Languages recognized by NFA with ϵ-transitions may also be embedded in such Eilenberg machines but they would not necessarily satisfy the nœtherian condition since there may be cycles of ϵ-transitions. Conversely a NFA with no such cycle is a finite Eilenberg machine.*

2.2 Rational transducers and finite Eilenberg machines

Let Σ and Γ be two finite alphabets. The empty word ϵ will denote both empty words for Σ^* and for Γ^*. We consider here the monoid $\Sigma^* \times \Gamma^*$ with its traditional concatenation as associative product and the pair (ϵ, ϵ) as unit element. A rational transducer from Σ^* to Γ^* is a monoid automaton $\mathcal{A}$ over $\Sigma^* \times \Gamma^*$ such that its type $\Phi_{\mathcal{A}} \subseteq (\Sigma \times \epsilon) \cup (\epsilon \times \Gamma)$. The subset of pairs of words from $|\mathcal{A}|$, the behavior of $\mathcal{A}$, defines a relation which belongs to the subclass of *rational relations*. A transducer defines a relation which solves the three following problems:

1. *Recognition* Given a couple of words (w, w') of $\Sigma^* \times \Gamma^*$, does (w, w') belong to $|\mathcal{A}|$.
2. *Synthesis* Given a word w in Σ^* compute the set $|\mathcal{A}|(w)$ of words from Γ^*.
3. *Analysis* Given a word w in Γ^* compute the set $|\mathcal{A}|^{-1}(w)$ of words from Σ^*.

Let us now provide the relabelling of the transducer $\mathcal{A} = (Q, \delta, I, T)$ for solving the *recognition* problem. The morphism α is completely defined on elements of $\Phi_{\mathcal{A}}$ with $\alpha(\sigma, \epsilon) = L_\sigma^{-1} \times id_{\Gamma^*}$ and $\alpha(\epsilon, \gamma) = id_{\Sigma^*} \times L_\gamma^{-1}$. The machine $\mathcal{M} = \alpha(\mathcal{A})$ is nœtherian since each transition of $\mathcal{M}$ consumes either

a letter from a word of Σ^* or a word of Γ^*. We complete the definition of the machine with the set $\mathcal{D}_- = \Sigma^* \times \Gamma^*$, the input relation $\phi_- = id_{\Sigma^* \times \Gamma^*}$, the set $\mathcal{D}_+ = \mathbb{B}$ and output relation ϕ_+ with $\phi_+(\epsilon, \epsilon) = \top$ or $\bot$ otherwise. Thus we have

$$(\phi_- \circ ||\mathcal{M}|| \circ \phi_+)^{-1}(\top) = |\mathcal{A}|.$$

This result concerning the *recognition* problem for rational relations is similar to the one concerning the word problem of NFA.

We now consider the relabelling for the *synthesis* problem. The morphism α is defined on $\Phi_{\mathcal{A}}$ such that $\alpha(\sigma, \epsilon) = L_\sigma^{-1} \times id_{\Gamma^*}$ and $\alpha(\epsilon, \gamma) = id_{\Sigma^*} \times R_\gamma$. This encoding is complete when we consider the Eilenberg machine $\mathcal{M} = \alpha(\mathcal{A})$ with the following interface: $\mathcal{D}_- = \Sigma^*$, $\mathcal{D}_+ = \Gamma^*$, the input relation $\phi_- = \{ (w, (w, \epsilon)) \mid w \in \Sigma^* \}$ and the output relation $\phi_+ = \{ ((\epsilon, w'), w') \mid w' \in \Gamma^* \}$. Then the relation described by this machine is:

$$(\phi_- \circ ||\mathcal{M}|| \circ \phi_+) = |\mathcal{A}| \ .$$

Such an Eilenberg machine is trivially proved to be locally finite. But in this case the Eilenberg machine $\alpha(\mathcal{A})$ might not always be nœtherian: for example, a cycle in the machine labelled only by relations of the form $id_{\Sigma^*} \times R_\gamma$ generates possibly infinite traces.

Nevertheless such configurations could not happen for length-preserving relations implemented by resynchronized transducers of type $\Sigma \times \Gamma$ which are compiled into finite Eilenberg machines. We have furthermore:

Proposition 2.1. *For any rational relation implemented by a trimmed rational transducer $\mathcal{A}$ of type $\Phi_{\mathcal{A}} \subseteq (\Sigma \times \epsilon) \cup (\epsilon \times \Gamma)$, the following conditions are equivalent:*

1. *For all $w \in \Sigma^*$ the set $|\mathcal{A}|(w)$ is finite.*
2. *There is no cycle of type $(\epsilon \times \Gamma)$ in $\mathcal{A}$.*
3. *The machine $\mathcal{M} = \alpha(\mathcal{A})$ is nœtherian.*

Proof.

(1) ⇒ (2) By contraposition. Assume that there exists such a cycle at some state q. The transducer being trimmed the state q is accessible and coaccessible. Thus there exists a word w_1 of Σ^* defining a path leading to q from an initial state and there exists a word w_2 of Σ^* defining a path leaving from q and arriving on a terminal state. We conclude that $|\mathcal{A}|(w_1 w_2)$ is infinite. Thus (1) is false.

(2) ⇒ (3) Assume the proposition (2) true. Since $\mathcal{A}$ is a finite automaton there exists an integer n bounding paths with labels of type $(\epsilon \times \Gamma)$. Let w be a word of Σ^* and let i be the length of w then the length of any trace beginning with (w, w') for any $w' \in \Gamma^*$ is at most $n \times i$. Thus $\mathcal{M}$ is nœtherian.

(3) ⇒ (1) From corollary 1.1. □

The proposition 2.1 shows that for the particular case of rational transducers, the implication of corollary 1.1 becomes an equivalence. This result is to be related to *positive rational relations* for which Eilenberg gave a resynchronization procedure.

The *analysis* problem might be solved similarly to the *synthesis* using R_σ instead of L_σ^{-1} and L_γ^{-1} instead of R_γ.

2.3 Applications to computational linguistics

Automata, transducers and more generally finite state machines are a popular technology for solving many computational linguistics problems [9]. We believe that Eilenberg machines are an efficient tool for this purpose. In fact, our restriction of *finite Eilenberg machines* is the formalism underlying the works concerning general morphological and phonetical modelizations [5, 6] which have been applied to the Sanskrit language modelization. Furthermore this application needs the modularity of such Eilenberg machine as sketched in the remark 1.1. In the following we provide algorithms that simulate in a complete fashion any finite Eilenberg machine. They are given with a complete formalization.

3 Streams representing relations

We consider now that the data set $\mathcal{D}$ is representable as an abstract ML datatype. We recall that ***unit*** is the singleton ML datatype containing the unique value denoted *()*. In our implementation we will use streams which are objects for enumerating on demand. In ML notation stream values are encoded with the following type parametrized with $\mathcal{D}$:

```
type stream D = | EOS
                | Stream of D * (delay D)
and delay D = unit -> stream D ;
```

A stream value is either the empty stream *EOS* ("*End of Stream*") for encoding the empty enumeration or else a value *Stream d del* that provides the new element *d* of the enumeration and a value *del* as a delayed computation of the rest of the enumeration. Since ML computes with the restriction of λ-calculus to weak reduction, a value of type *delay* $\mathcal{D}$ such as *del* is delayed because it is a functional value. This well known technique permits computation on demand. Note that this technique would not apply in a programming language evaluating inside a function body (strong reduction in λ-calculus terminology).

ML being a Turing-complete programming language, not all ML functions terminate and since our streams contain function values we shall restrict their computational power:

Definition 3.1.

1. *The ML function f:* **unit** → α *is said to be total iff the evaluation of f ()* *terminates (yielding a value of type* α*).*

2. *The ML stream str : stream $\mathcal{D}$ is said to be progressive iff*
   - *either str is EOS*
   - *or else str is of the form Stream d f with f total, and f () is progressive.*

We define the *head* function hd from non-empty streams to data defined as follows: $hd(Stream\ d\ del) = d$. We define also the *tail* function tl from streams to streams as $tl(EOS) = EOS$ and $tl(Stream\ d\ del) = del()$. Let n be an integer, we introduce the function tl^n that iterates the tl function n times:

$$\begin{cases} tl^0(str) & = & str \\ tl^{n+1}(str) & = & tl^n(tl(str)) \end{cases} \tag{1}$$

We introduce the predicate $InStream(d, str)$ that checks whether a data d appears in the stream str:

Definition 3.2. *$InStream(d, str)$ is true iff there exists an integer n such that $(tl^n(str))$ is non-empty and $hd(tl^n(str)) = d$.*

Definition 3.3. *A progressive stream str is finite iff there exists an integer n such that $tl^n(str) = EOS$.*

The *length* of a finite stream str, written $|str|$, is defined inductively as follows:

$$\begin{cases} |EOS| & = & 0 \\ |Stream\ d\ del| & = & 1 + |del()| \end{cases} \tag{2}$$

All finite streams of positive length end with a value of type *delay* $\mathcal{D}$ that associates to the unit element *()* the *EOS* stream announcing the end of the enumeration, typically:

```
value delay_eos = fun () → EOS;
```

We consider now relations of $R_{\mathcal{D}}$ representable as ML functions of the following type:

```
type relation 'D = 'D → stream 'D ;
```

That is, if a relation *rel* of type *relation* $\mathcal{D}$ corresponds to a relation ρ of $R_{\mathcal{D}}$ then:

$$\forall d\ d' \in \mathcal{D},\ d' \in \rho(d) \Leftrightarrow InStream(d', rel\ d).$$

In the following we will use this technique only for representing locally finite relations. That is, relations are encoded using finite streams which are *progressive* by definition 3.3. From now on, we shall assume that our Eilenberg machines are *effective* in the sense that their data domain $\mathcal{D}$ are implemented as an ML datatype and that every relation used in their labeling is progressive.

4 A reactive engine for finite Eilenberg machines

We provide an implementation for the simulation of finite Eilenberg machines using higher-order recursive definitions. Algorithms are presented using ML notations which are directly executable in the OCaml programming language [7]. An essential feature of our formal notations is to possibly compose parametrized modules called functors. Algorithms are variants of the *reactive engine* [5]. They are presented completely using only a dozen of elegant definitions.

4.1 The reactive engine

Let $\mathcal{M} = (Q, \delta, I, T)$ be a $\mathcal{D}$-machine. We specify $\mathcal{M}$ as a module with the following signature:

```
module type Kernel = sig
  type 𝒟 ;
  type state;
  value transition: state → list (relation 𝒟 × state);
  value initial: list state;
  value terminal: state → bool;
end;
```

The type parameter *state* encodes the set Q, the function *transition* encodes the function δ, the value *initial* encodes the initial states I as a list and the function *terminal* encodes terminal states T seen as a predicate.

We aim at providing the algorithms implementing the characteristic relation of $\mathcal{M}$. For this purpose we use a functor that is a module parametrized by a *Kernel* machine. We call *Engine* this functor declared as the following:

```
module Engine (M: Kernel) = struct
  open M;
  ...(* body *)...
end;
```

Firstly, the body of the functor contains type declarations:

```
type choice = list (relation 𝒟 × state);
type backtrack =
  | Advance of 𝒟 × state
  | Choose of 𝒟 × state × choice × (delay 𝒟) × state
  ;
type resumption = list backtrack;
```

The type *choice* is an abbreviation for the list of transitions of the machine as used in the machine *M*. Eilenberg machines are possibly non-deterministic and need thus a backtracking mechanism for their implementation. Values of type *backtrack* allow to save the multiple choices due to the non-deterministic nature of the machine. The enumerating procedure will stack such backtrack values in a resumption of type *resumption*.

Secondly, the engine contains four functions. Three of them are internal and mutually recursive: *react*, *choose* and *continue*. They perform the non-deterministic search enriching the resumption as the computation goes on. The

computation is performed in a depth-first search manner stacking the transition choices and streams within backtrack values in the resumption:

```
(* react: D → state → resumption → stream D *)
value rec react d q res =
  let ch = transition q in
  if terminal q
    then Stream d (fun () → choose d q ch res) (* Solution found *)
    else choose d q ch res

(* choose: D → state → choice → resumption → stream D
*)
and choose d q ch res =
  match ch with
  | [] → continue res
  | (rel, q') :: rest →
    match (rel d) with
    | EOS → choose d q rest res
    | Stream d' del →
      react d' q' (Choose(d,q,rest,del,q') :: res)

(* continue: resumption → stream D *)
and continue res =
  match res with
  | [] → EOS
  | (Advance(d,q) :: rest) → react d q rest
  | (Choose(d,q,ch,del,q') :: rest) →
    match (del ()) with
    | EOS → choose d q ch rest
    | Stream d' del' →
      react d' q' (Choose(d,q,ch,del',q') :: rest)
;
```

The function *react* checks whether the state is terminal and then provides an element of the stream delaying the rest of the exploration calling to the function *choose*. This function *choose* performs the non-deterministic search over transitions, choosing them in the natural order induced by the ***list*** data structure. The function *continue* manages the backtracking mechanism and the enumeration of finite streams of relations, it always chooses to backtrack on the last pushed value in the resumption. Remark that these three mutually recursive functions do not use any side effect and are written in a pure functional style completely tail-recursive using the resumption as a continuation mechanism.

The machine $\mathcal{M}$ implemented as a module M has its characteristic relation $||\mathcal{M}||$ simulated by the following function:

```
(* characteristic_relation: relation D *)
value characteristic_relation d =
  let rec init_res l acc =
    match l with
    | [] → acc
    | (q :: rest) → init_res rest (Advance(d,q) :: acc)
```

```
in continue (init_res initial []);
```

The function *characteristic_relation* first initializes the resumption with *Advance* backtrack values for each initial state and then call the function *continue* on it. We summarize the presented algorithms as follows: the machine $\mathcal{M}$ implemented as a module M has its characteristic relation $||\mathcal{M}||$ simulated by *Engine(M).characteristic_relation*, the function given by the instantiation of functor *Engine* with module M. We now provide the formalization with all arguments ensuring the correctness of our so-called reactive engine.

4.2 Formalization

The formalization is inspired by the original one for the reactive engine [5]. It uses the *multiset ordering* technique as presented by Dershowitz and Manna [3] to prove the termination of our algorithms. It also gives us a useful nœtherian induction principle for the soundness and completeness proofs.

We formalize the fact that data d and d' are in relation by the characteristic relation of $\mathcal{M}$ using the predicate $Solution(d, d')$ which is true iff there exists an initial state q and a terminal trace t beginning with cell (d, q) and ending with data d'.

Theorem 4.1. *If the machine $\mathcal{M}$ is finite then for all data d and d',*

$$InStream(d', characteristic_relation\ d) \Leftrightarrow Solution(d, d').$$

One direction of the equivalence of the theorem corresponds to the soundness of the algorithm and the other to its completeness.

Let us define $WellFormedBack(b)$, an invariant on backtrack values b met during the computation of the three mutually recursive functions inside an execution of the function *characteristic_relation* :

1. $WellFormedBack(Advance(d, q))$ for every data d and state q.
2. $WellFormedBack(Choose(d, q, ch, del, q'))$ when there exists a relation rel such that the following conditions are verified:
   - $(rel, q') \in (transition\ q)$,
   - $\forall d', InStream(d', del\ ()) \Rightarrow InStream(d', (rel\ d))$,
   - $\forall e, e \in ch \Rightarrow e \in (transition\ q)$.

We extend the property $WellFormedBack$ of backtrack values to resumptions by:

$$WellFormedRes(res) = (\forall b, b \in res \Rightarrow WellFormedBack(b)).$$

Before proving the theorem 4.1 we prove the termination of the algorithms. The machine $\mathcal{M}$ being assumed nœtherian, cells are partially-ordered with the edge relation and this ordering is nœtherian. Consider triples of the form $\langle c, n_1, n_2 \rangle$ with c being a cell and the other parameters n_1 and n_2 integers. Such triples together with extension of lexical ordering (using the traditional ordering on

natural numbers) form also a nœtherian partial-ordering. In the following we shall use the extension on multisets of such triples and its multiset-ordering [3].

Now we shall define a function χ which associates a multiset of triples to each function call of *react*, *choose* and *continue*. Proving that the multiset decreases along recursive calls shows that the functions terminate.

Definition 4.1. *If res is a resumption we define* $\chi(\text{res})$ *as the multiset of all* $\chi(\text{back})$*, for back a backtrack value in res, where*

$$\chi(Choose(d, q, ch, del, q')) = \langle (d, q), |ch|, |del()| + 1 \rangle$$

$$\chi(Advance(d, q)) = \langle (d, q), \kappa(q) + 1, 0 \rangle$$

with $\kappa(q) = |transition\ q|+1$*. We now associate such multisets to every function invocation react, choose and continue:*

$$\chi(react\ d\ q\ res) = \{\langle (d, q), \kappa(q), 0 \rangle\} \oplus \chi(res)$$

$$\chi(choose\ d\ q\ ch\ res) = \{\langle (d, q), |ch|, 0 \rangle\} \oplus \chi(res)$$

$$\chi(continue\ res) = \chi(res)$$

with $\oplus$ *being the multiset union.*

Using the function χ we have the following proposition:

Proposition 4.1. *If the machine* $\mathcal{M}$ *is finite, for all res value of type resumption, if* $WellFormedRes(res)$ *then continue res returns a stream value which is either EOS or (Stream d' del) for some data d' and delay del such that* $\chi(res) \gg \chi(del())$*.*

Proof. We first appropriately strenghten this proposition for functions *react* and *choose*. The proof is by simultaneous nœtherian induction over the multiset ordering computed by χ. Inspecting all the branches of the mutually recursive functions *react*, *choose* and *continue*, it is easy to see that χ decreases. □

Corollary 4.1 (Termination)**.** *If the machine* $\mathcal{M}$ *is finite, for all data d the execution of function characteristic_relation applied to d returns a finite (progressive) stream.* □

Now we consider the predicate $PartSol(c,\ d')$ true iff there exists a terminal trace t beginning with cell c and ending with data d'.

We extend this predicate on *choice* values as a $PartSolChoice$ predicate defined as the following: $PartSolChoice(d, ch, d')$ is true if and only if there exists a relation *rel* and a state q_1 with (rel, q_1) in *ch* and such that there exists a data d_1 in the stream *rel d* and such that $PartSol((d_1, q_1),\ q')$ is true.

We define the same kind of predicate for backtrack values called $PartSolBack$:

1. $PartSolBack(Advance(d, q),\ d')$ iff there exists a terminal trace beginning with (d, q) and ending with data d'.

2. $PartSolBack(Choose(d, q, ch, del, q_1),\ d')$ iff $PartSolChoice(d, ch, d')$ is true or else there exists a data d_1 in the stream *del ()* with $PartSol((d_1, q_1), d')$ being true.

We extend the property $PartSolBack$ of backtrack values to resumptions by:

$$PartSolRes(res,\ d') = (\exists b,\ b \in res\ \wedge\ PartSolBack(b,\ d')).$$

Now we may formulate the three following soundness and completeness lemmas for functions *react*, *choose* and *continue*:

Lemma 4.1. *If the machine $\mathcal{M}$ is finite then the three following properties are verified:*

1. $\forall\ d\ q\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $InStream(d', react\ d\ q\ res) \Leftrightarrow PartSol((d, q),\ d') \vee PartSolRes(res, d')$,
2. $\forall\ d\ q\ ch\ res\ d',\ WellFormedRes(res) \Rightarrow ch \subseteq (transition\ q) \Rightarrow$
   $InStream(d', choose\ d\ q\ ch\ res) \Leftrightarrow$
   $PartSolChoice(d, ch, d')\ \vee\ PartSolRes(res, d')$,
3. $\forall\ res\ d',\ WellFormedRes(res) \Rightarrow$
   $InStream(d', continue\ res) \Leftrightarrow PartSolRes(res, d')$.

Proof. The proof is a routine case analysis by simultaneous nœtherian induction over χ. □

Proof of Theorem 4.1. Let d be a data, the execution of *characteristic_relation* applied to d leads to the execution of *continue* applied to a resumption value $res = (init_res\ initial\ [])$. This resumption res is proved to be composed of backtrack values $Advance(d, q)$ with state q being necessarily an initial state. Also for such a resumption res we have $WellFormedRes(res)$. Applying the third case of the lemma 4.1 we obtain

$$InStream(d', continue\ res) \Leftrightarrow PartSolRes(res, d').$$

Also with the considered resumption res, for any data d' we have

$$PartSolRes(res, d') \Leftrightarrow Solution(d, d')$$

because states inside the resumption are initial. Then we conclude by transitivity of the two equivalences. □

Termination, soundness and completeness ensure that the function *characteristic_relation* implements a locally finite relation as a *relation* $\mathcal{D}$. It illustrates the proposition 1.1 and ensures the correctness of the reactive engine that simulates any finite Eilenberg machine.

5 Conclusion

Eilenberg machines provide a powerful and elegant framework for simulating specifications presented as finite automata variants. Eilenberg gave easy encodings into machines of formalisms at various levels of the Chomsky hierarchy. We have shown that our subclass of *finite Eilenberg machines* is large enough to support NFA and many transducers. Other examples might be given since our restriction keeps most of the generality brought by original Eilenberg machines. Our machines are not restricted to treatments for the rational level of the Chomsky hierarchy. This particular point makes us believe that finite Eilenberg machines have applications to computational linguistics. In fact they are already efficient for explaining recognition or transduction problems that manipulate two levels of finite state formalisms such as explained in [6] for the modelization of the Sanskrit language. This multi-level ability is a feature of Eilenberg machines that we call modularity. For this purpose implementations need to be lazy. We anticipate future works in this spirit providing lazy algorithms. Our small but efficient *reactive engine* computes lazily the simulation of any finite Eilenberg machines. Our methodology using higher-order recursive definitions of functional programming language leads to formal proofs amenable to a complete formalization using higher-order logic. Such a formal development is available in the companion paper [8] using the proof assistant for Coq [2].

Acknowledgments

Gérard Huet and Jean-Baptiste Tristan took an essential part in the elaboration of this paper; I thank them for their intensive participation.

References

[1] Theory of X-machines. `http://www.x-machines.com`.

[2] The Coq proof assistant. Software and documentation available on the Web, `http://coq.inria.fr/`, 1995–2007.

[3] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[4] Samuel Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.

[5] Gérard Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional programming*, 15, 2005.

[6] Gérard Huet and Benoît Razet. The reactive engine for modular transducers. In Jean-Pierre Jouannaud Kokichi Futatsugi and José Meseguer,

editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006.

[7] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr/`, 1996–2006.

[8] Benoît Razet. Simulating Eilenberg machines with a reactive engine: Formal specification, proof and program extraction. Research Report 6487, INRIA, 02 2008.

[9] Emmanuel Roche and Yves Schabes. *Finite-state language processing*. MIT press, 1997.

[10] Jacques Sakarovitch. Eléments de théorie des automates. *Vuibert, Paris*, 2003.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399